Concurrency-Informed Orchestration for Serverless Functions

Qichang Liu
University of Virginia
Charlottesville, Virginia, USA
nzc5ve@virginia.edu

Yue Cheng
University of Virginia
Charlottesville, Virginia, USA
mrz7dp@virginia.edu

Haiying Shen
University of Virginia
Charlottesville, Virginia, USA
hs6ms@virginia.edu

Ao Wang
Alibaba Group
Hangzhou, Zhejiang, China
shenlan.wa@alibaba-inc.com

Bharathan Balaji
Amazon
Seattle, Washington, USA
bhabalaj@amazon.com

Abstract

Cold start delays are a main pain point for today's FaaS (Function-as-a-Service) platforms. A widely used mitigation strategy is keeping recently invoked function containers alive in memory to enable warm starts with minimal overhead. This paper identifies new challenges that state-of-the-art FaaS keep-alive policies neglect. These challenges are caused by concurrent function invocations, a common FaaS workload behavior. First, concurrent requests present a trade-off between reusing busy containers (delayed warm starts) versus cold-starting containers. Second, concurrent requests cause imbalanced evictions of containers that will be reused shortly thereafter. To tackle the challenges, we propose a novel serverless function container orchestration algorithm called CIDRE. CIDRE makes informed decisions to speculatively choose between a delayed warm start and a cold start under concurrency-driven function scaling. CIDRE uses both fine-grained container-level and coarse-grained concurrency information to make balanced eviction decisions. We evaluate CIDRE extensively using two production FaaS workloads. Results show that CIDRE reduces the cold start ratio and the average invocation overhead by up to 75.1% and 39.3% compared to state-of-the-art function keep-alive policies.

CCS Concepts: • **Computer systems organization** → Architectures; Distributed architectures; **Cloud computing**;

Keywords: Cloud Computing; Serverless Computing; Function-as-a-Service; Autoscaling; Container Orchestration; Caching

ACM Reference Format:
Qichang Liu, Yue Cheng, Haiying Shen, Ao Wang, and Bharathan Balaji. 2025. Concurrency-Informed Orchestration for Serverless Functions. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3676641.3716253

ASPLOS '25, Rotterdam, Netherlands

ACM ISBN 979-8-4007-1079-7/2025/03
https://doi.org/10.1145/3676641.3716253

1 Introduction

Serverless computing enables building and scaling applications by breaking monolithic applications into fine-grained functions [32]. Developers supply function logic while the serverless provider performs the tasks of provisioning, scaling, and caching backend servers on which the functions execute [28]. Serverless computing has become a well-established paradigm, with FaaS (Function-as-a-Service) offerings widely adopted in commercial platforms (e.g., AWS Lambda [10], Azure Functions [11]) and supported by open-source frameworks (e.g., OpenWhisk [8], Knative [2], OpenLambda [44]).

FaaS workloads are fundamentally different from traditional computing services. Serverless functions are ephemeral and typically have short execution time, ranging from milliseconds (ms) to seconds [49]. *The transient nature of function execution poses significant challenges to FaaS provisioning decisions.* Serverless functions are executed in isolated sandbox environments such as virtual machines (VMs) [4] and/or containers [43, 57]. A function *startup* process involves downloading and installing the environment (OS image, language runtime, and dependencies) before the function code can be executed. This "*cold start penalty*" can be substantial compared to function execution time. It may experience delays up to two orders of magnitude longer than a warm start [23, 52] before function execution begins, ultimately affecting end-to-end application performance and user experience.

Another challenge is that FaaS workloads are highly concurrent [6, 18, 19, 53]. *It is not uncommon for a single function to spike up to thousands of requests concurrently* [34]. To sustain the high concurrency, FaaS platforms provision many containers replicated from the same function deployment [16, 53]. This workload behavior further exacerbates the impact of cold starts on *function invocation overhead.*

A straightforward approach to alleviating cold starts is to keep invoked function containers alive in host memory for a configurable period of time [27, 49, 54, 55, 63]. By doing so,

subsequent function invocation requests can directly execute within the already-initialized containers, thereby reducing startup latency. Caching is one of the most well-studied problems [3, 12, 25, 38, 39] in online computation. State-of-the-art function keep-alive policies such as FaasCache [27] and OpenFaaS [42] treat keep-alive as a traditional caching problem. Specifically, a caching-based keep-alive policy treats a warm function as a cached object, a warm function execution as a cache hit, and reclaiming a warm function as evicting an object from the cache, with objectives of maximizing warm starts (hits) and minimizing execution delay.

In this paper, by analyzing production FaaS workloads, we identify challenges that the state-of-the-art (SOTA) function keep-alive policies neglect. These challenges are caused by concurrent function invocations, a common FaaS workload behavior. Specifically, we make **two key observations**:

- **Delayed warm starts:** FaaS concurrency introduces a new tradeoff between reusing a busy warm container (i.e., a delayed warm start that incurs a queuing delay) versus creating a new container (that incurs a cold start delay). Existing FaaS platforms neglect the subtle opportunity to exploit busy warm containers in the face of concurrent requests. This oversight leads to excessive cold starts and more containers created, affecting the overall performance.
- **Imbalanced evictions:** FaaS concurrency causes imbalanced, bulk evictions of containers that might be soon reused. Traditional caching-based keep-alive policies fail to accurately capture fine-grained container-level and coarse-grained function-level behaviors. Ill-suited priorities suffer suboptimal ordering of warm containers, resulting in less-informed eviction decisions and ultimately contributing to higher function invocation overhead.

To address the challenges, based on the observations, we propose a novel serverless function container orchestration algorithm called concurrency-informed delayed reuse and eviction (CIDRE). CIDRE synergies two innovative and effective concurrency-aware orchestration policies listed below to inform function container orchestration decisions throughout the entire lifecycle, from function scaling to eviction.

- **A better function scaling policy that maximizes the utilization of warm containers.** Informed by an intelligent speculative scaling policy, CIDRE reuses busy warm containers whenever it can reduce invocation delays. By intelligently deciding whether to (1) reuse a busy warm container, (2) issue a cold start, or (3) do both simultaneously—a technique we call conditional speculative scaling—CIDRE optimizes function scaling to minimize functions' invocation overhead and reduce the number of cold starts.
- **A better cache eviction policy that minimizes unnecessary warm container evictions.** CIDRE uses both fine-grained container-level information and coarse-grained function-level information to determine which warm containers should be replaced with cold starts to avoid evicting containers that will be reused shortly.

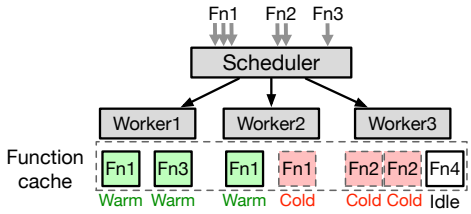


Figure 1. Function invocation process. In this example, four function containers are cached, with three being actively used and one in the idle state. Concurrent invocation requests to Function 1 (Fn1) result in two warm starts (hits) and one cold start (miss), while concurrent invocation requests to Fn2 see two cold starts.

We make the following **contributions** in this paper.

- We conduct comprehensive analyses on production FaaS workloads and identify the challenges and the limitations of traditional-caching-based function keep-alive policies to address the challenges.
- We present a new tradeoff between delayed warm starts and cold starts, and the notion of balanced evictions.
- We propose CIDRE with two effective techniques: a speculative scaling policy and a concurrency-informed priority eviction policy. To minimize invocation overhead, CIDRE reuses busy containers whenever desirable during function scaling and uses comprehensive workload knowledge to inform eviction decision.
- We implement CIDRE in OpenLambda [30, 44] and evaluate CIDRE using production FaaS traces from Azure Functions [49] and Alibaba Cloud Function Compute (FC) [1]. Results show that CIDRE reduces the cold start ratio and the average invocation overhead by up to 75.1% and 39.3% compared to SOTA FaaS keep-alive solutions.

Real-World Impact and Artifact Availability. CIDRE's speculative scaling has received adoption and is deployed in production at Alibaba Cloud FC. CIDRE is open-sourced and available at: https://github.com/nzc5ve/cidre_asplos25.

2 Background and Motivation

2.1 Overview of FaaS

Function Deployment and Invocation. A user deploys a serverless function by pushing function code to a function registry. A deployed function can be invoked, e.g., through an HTTP URL. Each invoked function runs in a sandbox environment, e.g., a container [22, 29] or a VM [4]. Without loss of generality, we assume *containers* as the underlying sandbox technique for function execution and isolation.

Function Lifecycle. A FaaS platform typically consists of a scheduler and a cluster of workers (see Figure 1). Function invocation requests are forwarded to a cluster of servers through the scheduler. Each server in the cluster runs a worker which is responsible for managing the lifecycle of containers hosted on that server. Upon receiving multiple concurrent invocation requests, the worker starts multiple containers of the function to serve the requests (i.e.,

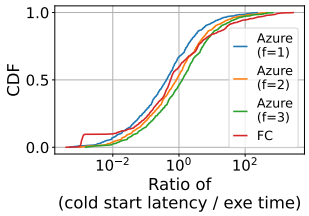


Figure 2. Distribution of the cold start latency to function execution time ratio.

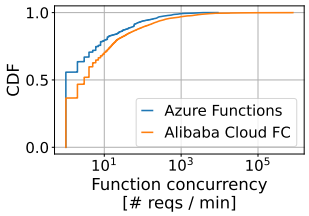


Figure 3. Function concurrency CDFs (each point in the curve: *reqs/min* of a function).

concurrency-driven container scaling). The *cold start* process involves downloading and installing the container image, language runtime, and dependencies of the function before the function code can be executed. When a function finishes execution, the worker may choose to *keep its containers alive* in the server's memory for a short period of time. A *warm start* incurs a much smaller invocation overhead, as it reuses an already provisioned container for request execution.

2.2 Real-World Workload Analysis

Quantifying Cold Start Overhead. We sampled 1, 267 cold-started function invocations from a production FaaS workload collected in Alibaba Cloud FC. In the FC production environment, a *cold start* refers to the process of starting a new container. This includes downloading/loading the container image, initializing the language runtime, loading the function code and user data, and establishing any network/DB connections. A *warm start* occurs when a function request is executed in an available container. A warm start skips image loading and runtime initialization but *may still involve a warmup phase*. A *warmup phase* during a warm start can include tasks like JIT compilation, ML model downloading, and establishing connections, which are request-driven.

Figure 2 shows the ratio of cold start latency to execution time. Among all cold starts, 40.4% of requests have a ratio greater than 1 with non-trivial invocation overhead. We also randomly sampled 750 unique functions from Day 1 of the Azure Functions workload [49]. The full trace contains function invocation requests of 82, 375 unique functions spanning 14 days. Given that the original dataset lacks information about cold starts, we calculated the estimated cold start latency by applying one of three scaling factors, $1ms/MB$, $2ms/MB$, and $3ms/MB$, to the average allocated memory [52]. We found that the estimated cold start overhead follows the same distribution as that of the FC trace, suggesting a huge impact of cold start cost on the end-to-end function performance and user experience.

Quantifying Function-level Concurrency. Concurrent invocation requests are common in FaaS applications such as stateless image processing [48] and burst-parallel, stateful workflow processing [6, 26, 31]. Concurrent requests target the same function and are often issued at roughly the same time, leading to the creation of multiple containers

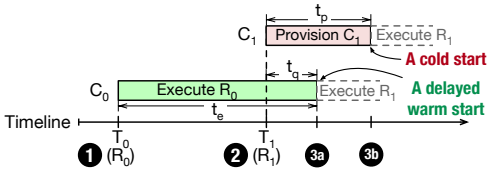


Figure 4. Concurrent invocation requests (R_0 and R_1) to the same function F present a tradeoff. 1: R_0 arrives at timestamp T_0 and a warm container C_0 is already kept alive to directly serve R_0; it takes R_0 a duration of t_e to execute. 2: A concurrent request R_1 to the same function arrives at T_1, where $T_1 - T_0 < t_e$. 3: Serving R_1 involves a decision making. 3a: reusing the busy container C_0 will incur a queuing delay of t_q; 3b: provisioning a new container C_1 to serve R_1 will incur a cold start latency of t_p. Then, the optimal decision is to reuse C_0 as $t_q < t_p$.

in one or multiple servers (Fn1 in Figure 1). To quantify this, we measured the request concurrency of a 30-minute FC workload. As shown in Figure 3, the {90^{th}%-ile, 99^{th}%-ile} concurrency is {120, 4,482}, respectively, suggesting that real-world production FaaS workloads are highly concurrent. The sampled set of 750 functions in the Azure Functions workload exhibit a similar distribution pattern, although the concurrency level of Azure Functions is slightly lower than that of Alibaba Cloud FC. Since the Azure traces only provide coarse-grained, minute-level concurrency information, we modeled second-level concurrency by following the concurrency distribution from FC traces.

2.3 Problems with Concurrent Invocation Requests

New Tradeoff. We find that concurrent invocations to the same function introduce a new tradeoff as illustrated in Figure 4: reusing a busy container that is actively serving a request incurs a queuing delay, while provisioning a new container introduces a cold start latency. In the example shown in Figure 4, invocation request R_1 would have waited for an extra duration of $t_p - t_q$ if the scheduler decides to provision a new container C_1 and waits for C_1 to be fully initialized before executing R_1.

Traditional Caching Models. A caching policy is an algorithm designed to determine, given a cache size and a workload of object access requests, whether or not to admit a new object into the cache upon a miss, i.e., the admission policy, and if so, which object to evict, i.e., the eviction or replacement policy. Caching policy targets to achieve a particular objective, e.g., to minimize the miss ratio or to maximize the hit ratio. Existing FaaS platforms model the function keep-alive problem as a classic caching problem [27, 42].

Problems with Caching-based Keep-Alive. The new tradeoff subverts the expectation of traditional-caching-based keep-alive when it comes to *function invocation overhead*, i.e., the waiting time incurred before a function starts execution. Traditional caching models ignore the compound impact of function concurrency, and therefore, suffer from two main problems. The two problems are logically correlated and can significantly affect workload performance.

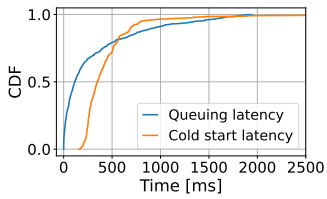


Figure 5. Tradeoffs of reusing busy warm containers vs. cold starts (Azure trace).

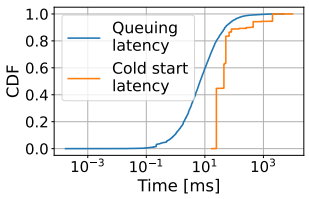


Figure 6. Tradeoffs of reusing busy warm containers vs. cold starts (FC trace).

- **Problem of Delayed Warm Starts:** Traditional caching policies such as GDSF (Greedy-Dual-Size-Frequency) [20] and LRU have only two states for each request, a cache hit or a cache miss. The new tradeoff under function concurrency introduces a new intermediate state between a true "hit" (i.e., a warm start) and a true "miss" (i.e., a cold start)—a *delayed hit* [9] (i.e., a *delayed warm start*, which reuses a busy warm container but waits for a queuing delay before the function can execute, as illustrated in Figure 4).
- **Problem of Imbalanced Eviction:** Concurrency causes another interesting problem: a function can have multiple containers kept alive; all containers associated with a function construct *an elastic, compound, logical object*, which can grow and shrink, driven by function concurrency. Traditional caching policies make independent eviction decisions by evicting the least important objects but are compound-object-oblivious. As such, blindly evicting the least important functions may lead to imbalanced eviction, thus affecting overall performance.

In the presence of function concurrency, the correlated problems of delayed warm starts and imbalanced eviction present a unique opportunity to further reduce function invocation overhead. During concurrency-driven container scaling, some requests could reuse a busy, warm container with shorter waiting time than the case if the requests were to be provisioned with new containers. Moreover, the FaaS scheduler should be intelligent enough to dynamically balance the cache space allocated for each compound function object. As a consequence of this gap, state-of-the-art caching-based keep-alive policies fail to minimize function invocation overhead, which we demonstrate next in §2.4.

2.4 What-If Analysis

In this section, we present a what-if analysis to quantify our identified tradeoff and to better understand its implications. All experiments throughout the paper were conducted using a warmed-up function cache.

Quantifying the Tradeoff. Our first what-if experiment analyzes what the cost and benefit would be if a GDSF-based FaasCache has the option to reuse a busy container. We replayed the 14.7 million function requests from the 750 sampled Azure functions (see Table 1) using a simulator developed by ourselves, which simulates a modified version of FaasCache. Since vanilla FaasCache does not reuse a busy warm container, we modified its policy so that, if a request triggers a cold start (i.e., no idle warm containers are available to serve this request), the modified policy will instead route this request to a busy warm container that has the shortest waiting time. This way, the modified policy avoids cold starts but enforces a queuing delay. Figure 5 quantifies this tradeoff. Interestingly, the two CDF curves cross at 464ms. Around 69.4% of requests would have experienced significantly shorter invocation delays (< 464 ms) if FaasCache had reused a busy warm container instead of creating a new container. On the other hand, there is a 30.6% possibility that having a cold start would be the optimal choice.

Table 1. Production workload statistics. AF: Azure Functions. FC: Alibaba Cloud Function Compute. Rps: requests per second. GBps: the aggregate memory size of all requests per second in GBs.

Trace	# invoke reqs	Rps (avg / min / max)	GBps (avg / min / max)
24h AF	14,704,439	170 / 90 / 683	38.6 / 19.2 / 154.6
30m AF	3,231,319	1,795 / 1,158 / 4,551	804.5 / 502.1 / 2,014.7
30m FC	2,745,241	1,525 / 894 / 2,980	773.1 / 188 / 2,767

We did the same experiment using the 30-minute FaaS trace collected from Alibaba Cloud FC (Table 1). Figure 6 reveals a different pattern: all cold start requests encounter lower invocation overhead if FaasCache had opted to reuse a busy warm container. This result suggests a larger opportunity space potentially exposed by this tradeoff.

What If Delayed Warm Start is Enabled? Next, we analyze the impact of reusing busy warm containers on function invocation overhead. In this what-if experiment, we varied the queue length of busy warm containers from 0 to 2 and ran the same Azure Functions workload trace with a modified FaasCache. A queue length of L means a FaasCache policy that: (1) allows up to L enqueued function requests on any busy warm container, and (2) only creates a new container when delayed warm start queues are filled up for all busy warm containers. An L set to 0 means the vanilla FaasCache policy, which always creates a new container if all warm containers of the requested function are busy, representing an extreme end in the tradeoff spectrum.

The *overhead ratio* of a request is the ratio of its wait time and the sum of its wait time and execution time (Overhead ratio = $\frac{\text{wait time}}{\text{wait time+exe time}}$). Figure 7 shows the overhead ratio averaged across all requests. Allowing each busy warm container to enqueue up to one outstanding request reduces the average overhead ratio by 9.3% compared to vanilla FaasCache. Increasing the queue length from 1 to 2 results in a higher average overhead ratio than vanilla FaasCache. One should note that this policy, albeit sub-optimal, still outperforms FaasCache when the queue length is set to 1.

Observation 1

• *For an incoming function request, the queuing delay on a reused, busy, warm container might be shorter than the cold start latency of creating a new container.*

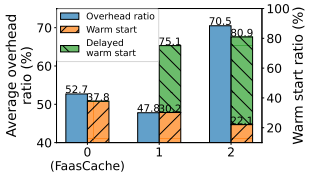


Figure 7. Impact of varying warm containers' queue length.

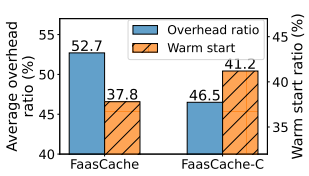


Figure 8. Impact of concurrency aware eviction.

• *A new policy is needed to make informed decisions about whether to choose a delayed warm start or a cold start.*

What If Concurrency-Aware Eviction is Enabled? In this experiment, we study the impact of concurrency-aware eviction on function invocation overhead. FaasCache adopts a GDSF-based keep-alive policy, which computes the priority of each container using the following equation:

$$Priority = Clock + Freq \times \frac{Cost}{Size} \quad (1)$$

where *Clock* captures the recency of the function; *Freq* is the aggregate number of invocations received by all cached containers of a function; *Cost* is the time required to provision the container; and *Size* denotes the memory footprint of the container. FaasCache evicts the containers with the lowest priorities. Since all containers of a function have the same *Cost*, *Size*, and *Freq*, FaasCache would evict the oldest and least-recently-used containers.

We modified FaasCache's eviction policy by incorporating a new metric K to represent concurrency and call it FaasCache-C:

$$Priority = Clock + Freq \times \frac{Cost}{Size \times K} \quad (2)$$

K denotes the number of warm containers currently cached for a function. With the updated policy, a function with more warm containers cached is more likely to be evicted, and the priority of a function tends to increase if more of its containers are evicted. Figure 8 shows the average overhead ratio of these two policies. Being concurrency-aware, FaasCache-C exhibits a 11.8% reduction in the average overhead ratio compared to vanilla FaasCache. Vanilla FaasCache tends to evict an entire function or a substantial portion of it, likely because these containers of victim functions happen to be clustered towards the lowest-priority end of the priority queue. In contrast, FaasCache-C leads to more balanced evictions with a 9% higher warm start ratio than vanilla FaasCache.

Observation 2

• *Traditional caching-based keep-alive policy that makes independent, container-level eviction decisions performs worse than a simple, concurrency-aware eviction policy.*
• *We need to find a more effective priority policy that can capture both container- and function-level behavior.*

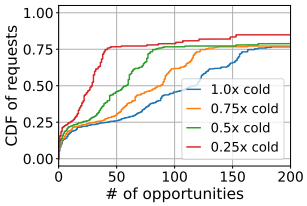


Figure 9. Impact of varying the cold start overhead.

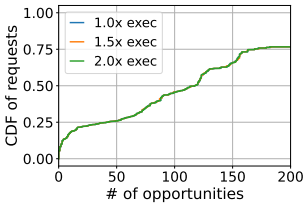


Figure 10. Impact of varying the execution time.

2.5 Quantifying Theoretical Opportunity Space

In this section, we present an in-depth trace analysis to quantify the opportunity space exposed by delayed warm starts.

Notations, Definition, and Analysis Methodology. We denote t_a as the arrival time of a newly arrived invocation request for function f, t_c as f's cold start overhead, and t_e as the function execution time. We define the opportunity space window as $[t_a, t_a + t_c]$. We analyzed all invocation requests in the 30-minute Azure Functions trace (Table 1). For each new request for a function f with an opportunity space window of $[t_a, t_a + t_c]$, we calculated the completion times $t_a + t_e$ for all other requests associated with f. We then identified and counted how many of these requests had completion times falling within the opportunity space window of the current new request. When analyzing the opportunity space of each newly arrived request, we make two assumptions. First, we assume that the new request always causes the creation of a new container, i.e., a cold start, The rationale behind this assumption is a what-if ananlysis: what if there is a cold start, then how many delayed warm start opportunities this request could get during the creation of the cold-started container. Second, we assume that all other requests associated with the same function in the trace have ideally zero invocation overhead. The rationale behind this assumption is that, unlike a simulation study, our theoretical analysis relies solely on the request information from the original trace and does not generate runtime information for each request, such as whether it experiences a cold start or a warm start. Therefore, we assume that all other requests, regardless of whether they fall within the opportunity space window, have ideally zero invocation overhead, that is, an ideal warm start without any extra overhead[1].

Analysis Results. Figure 9 shows the CDF of the number of delayed warm start opportunities with varied cold start overhead, specifically at 1.0×, 0.75×, 0.5×, and 0.25× the original cold start overhead t_c. As t_c decreases, the opportunity space window $[t_a, t_a + t_c]$ becomes smaller, reducing the number of delayed warm start opportunities. However, even with a 0.25× original cold start overhead, about 60% of requests still have more than 25 delayed warm start opportunities, making them likely to benefit from reduced queueing delays.

[1]We also tested the scenario where all other requests are assumed to be cold starts and the trend remained nearly identical.

Figure 10 depicts the CDF of the number of delayed warm start opportunities with varied execution time, specifically at 1.0×, 1.5×, and 2.0× the original execution time t_e. Interestingly, varying the execution time does not affect the opportunity space $[t_a, t_a + t_c]$. This is because changing the execution time only proportionally shifts the absolute positions of all request completion times, resulting in a uniform shift across the entire trace as all request execution times are adjusted simultaneously.

Observation 3

- *While varying the cold start overhead impacts the potential opportunity space of delayed warm starts, varying the execution time alone does not.*
- *Delayed warm starts exposes a big opportunity space to reduce latency compared to cold starts.*

2.6 Challenges of Exploiting the Tradeoff

One way to exploit the tradeoff is by accurately predicting the costs associated with delayed warm starts and cold starts to inform function scaling decisions. The cost of a function cold start is relatively predictable since it consistently accesses the same container image data across cold starts [52]. Predicting the cost of a delayed warm start, however, is challenging. The cost of a delayed warm start is the remaining execution time of the current function request that is being served by a busy warm container. The challenge is two-fold: (1) Different invocations of the same function may have variable execution time. (2) Keeping track of this cost requires fine-grained bookkeeping with extra overhead. A FaaS platform may have an enormous number of busy warm containers with different delayed warm start costs. Thus, finding the best busy warm container with the smallest cost is cost-prohibitive for large FaaS deployments.

We analyze the variance of function execution time for both the Azure and FC traces. We find that the majority of functions in both workloads have a marginally high variance of 25%—this accounts for 68% of functions for the Azure trace and 59% of functions for the FC trace. The observations indicate that using historical knowledge to estimate function execution time is error-prone. In fact, existing research suggests that the function execution time may be correlated to various factors such as input sizes [15, 37, 60] and function memory footprint [24, 47]. Therefore, in this work, we make a practical assumption that the execution time of a given function is volatile.

The aforementioned challenge requires us to find a solution that (1) can exploit the new tradeoff we identify in §2 to inform function scaling decisions and (2) effectively addresses the issue of function execution variance.

3 The CIDRE Orchestration Policy

Motivated by the observations from §2 and §2.6, we propose the CIDRE function orchestration policy.

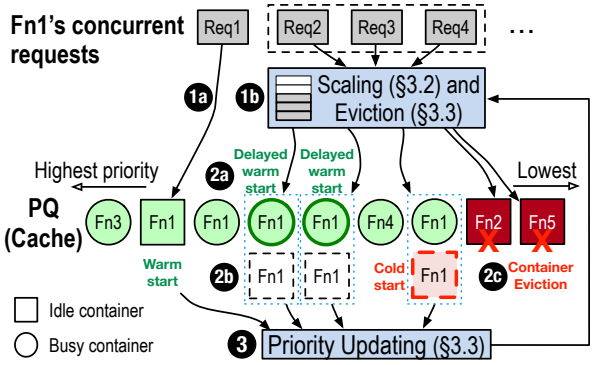


Figure 11. CIDRE architecture. CIDRE organizes all cached warm containers using a priority queue (PQ). In this example, CIDRE uses (conditional) speculative scaling to serve Req2-4 by speculatively choosing a busy warm container or a cold start, whichever has a shorter queueing delay; CIDRE replaces two idle warm containers of Fn2 and Fn5 with the lowest priority.

3.1 Design Overview

Figure 11 depicts the architecture of CIDRE and examples of delayed warm start and cold start paths. A detailed discussion about individual components appears in the next subsections, but the following gives a global picture of how CIDRE works. The main component of a CIDRE-managed function cache is a PQ structure, which sorts all busy and idle warm function containers by their priority values (§3.3). Unlike traditional caching models that process each request independently, CIDRE incorporates *function concurrency* in its orchestration policy spanning both function scaling and eviction. Invocation requests targeting a specific function may arrive in a burst, requiring the provisioning of multiple containers to serve concurrent requests.

In Step 1a, when a function's concurrent requests arrive, CIDRE first dispatches requests to available idle containers. In Step 1b, for the rest of requests for which CIDRE cannot find idle containers, CIDRE performs speculative scaling (SS) (§3.2) to determine whether to reuse a busy warm container or use a newly created container, given the current system state. The objective of SS is to minimize the invocation overhead incurred while waiting for container resources to become available. Step 1b occurs concurrently with Step 1a. CIDRE then performs the following two steps concurrently: CIDRE dispatches the outstanding requests to a queue managed by the speculative scheduler to wait for containers to become available (Step 2a) while provisioning new containers (Step 2b); CIDRE then executes function requests using containers that become available at the earliest. For example, if the first two busy warm containers of Fn1 become available sooner than the corresponding cold starts, CIDRE would execute Req2 and Req3 of Fn1 to these two just-vacant warm containers; then one cold start completes before the third busy warm container Fn1 finishes serving its current function request, therefore, CIDRE would

execute Req4 in the newly created container. We discuss an enhanced SS policy (conditional speculative scaling or CSS) that minimizes the cold start waste in §3.2.

CIDRE's CSS approach minimizes functions' invocation overhead with less cold starts. In existing FaaS platforms such as OpenLambda [30] and OpenWhisk [45], the function scheduler dispatches invocation requests to containers using a predefined scheduling policy such as round-robin, where the function workers (the PQ layer in Figure 11) are passively serving requests dispatched from the scheduler. This design results in higher invocation overhead and lower resource utilization. In contrast, CIDRE is *work-conserving* by serving requests whenever any busy/not-ready resources become available. In Step 2c, CIDRE *evicts* some warm containers from the lower-priority end of the PQ to provision new containers (§3.3). Note that Step 2c is concurrent with Step 2a and 2b. Lastly, in Step 3, CIDRE updates priorities for all containers touched during previous steps (§3.3 and §3.4).

3.2 Speculative Scaling

Basic Speculative Scaling. CIDRE uses a simple yet effective technique called speculative scaling to address the issue of function execution time variance. Instead of predicting the delayed warm start cost and the cold start cost, CIDRE speculatively chooses between a delayed warm start and a cold start. This strategy provisions new containers while monitoring the state of busy warm containers that are currently serving requests of the function. If any busy warm container finishes execution and becomes available, the scheduler dispatches the pending request to that vacant container without needing to wait for a new container to be fully created. If otherwise the provisioning process of a new container completes first, then the scheduler simply sends the request to the newly created container.

Basic SS (CIDRE_BSS or BSS) provides a worst-case performance guarantee: it guarantees that any function requests will experience an invocation overhead at least as good as that of a cold start. BSS achieves reduced invocation overhead without relying on sophisticated, often error-prone, cost modeling and prediction.

CIDRE_BSS has a drawback. It enforces a cold start for each speculative waiting action. The containers provisioned from these cold starts might become wasteful in hindsight if they will not be reused soon, or in the worst case, be evicted without being reused. Furthermore, cold starts evict existing warm containers that might be reused shortly afterward, causing cache thrashing and performance degradation. We present an enhancement to BSS next.

Conditional Speculative Scaling. While BSS minimizes the request waiting time, it might be wasteful to over-provision containers that will not be used in the near future. A newly provisioned container might stay idle without being invoked for a while, suggesting that: (1) there are enough warm containers to sustain the requests for that particular function, and (2) the last cold start brought by BSS is wasted. On the other hand, the newly provisioned container might be evicted before being invoked, suggesting that the current working set is around other functions. Worse, a wasted cold start causes the eviction of another function's container, affecting the performance of that function.

Algorithm 1 Conditional speculative scaling (CSS).

Input: Trigger to turn on/off basic SS: BSS; for the current request targeting Function F: estimated execution time T_e, cold start time T_p, last created container's idling time T_i, and last busy warm container's waiting time T_d.

```
1:  if BSS = True then                                   ▷ BSS has been enabled.
2:      if T_i > T_e then
3:          BSS ← False                                  ▷ Disable BSS.
4:          Perform a delayed warm start and update T_e and T_d
5:      else if T_i ≤ T_e then                           ▷ BSS path.
6:          if a busy warm container becomes available first then
7:              Perform a delayed warm start and update T_e and T_i
8:          else if the new container finishes provisioning first then
9:              Perform a cold start and update T_e
10: else                                                 ▷ BSS has been disabled.
11:     if T_d > T_p then
12:         BSS ← True                                   ▷ Re-enable BSS.
13:         if a busy warm container becomes available first then
14:             Perform a delayed warm start and update T_e and T_i
15:         else if the new container finishes provisioning first then
16:             Perform a cold start and update T_e
17:     else if T_d ≤ T_p then                           ▷ Keep BSS disabled.
18:         Perform a delayed warm start and update T_e and T_d
```

To address the problem, we propose an enhancement to BSS, called conditional speculative scaling or CSS. Rather than always provisioning a new container during the speculative wait, CSS performs *a cost-benefit analysis* to determine if it is worth creating a new container for the new request.

Algorithm 1 presents the logic of our CSS policy. Starting off, CIDRE performs BSS with both the delayed warm start path and cold start path enabled. CIDRE keeps track of the idling time of the last container that has been created via a cold start from the previous BSS process, defined as T_i. Specifically, T_i measures the duration between the time when the new container finishes provisioning to the time when it is reused. If T_i is longer than the expected execution time of that function T_e (defined as the median of all historical execution times of that function), it suggests that at least one busy warm container may become available during the idling period T_i (line 1-2). Thus, CSS determines that the previous cold start for that function is wasteful and could have been avoided. Then CIDRE disables the cold start path and will enforce this function to choose the delayed warm start path for all its upcoming invocation requests (line 3-4). CSS updates T_e and T_d (which is defined below) by incorporating the new requests (line 4). Otherwise, CSS determines that it might still be beneficial to do BSS (line 5-9).

CSS may toggle the cold start path back on if CIDRE predicts that none of the busy warm containers will become available within a short time. CSS makes this decision by

comparing two metrics: the duration that CIDRE waits to find an idle container since the last request arrives, defined as the delayed warm start cost, T_d, and the estimated container provisioning time obtained using the median of all historical cold start latency, defined as T_p. Assuming the cold start path is disabled for a function (line 10), CSS will re-enable the cold start path if T_d is longer than T_p. A longer T_d suggests that the cost of a delayed warm start is greater than that of a cold start, and thus, the system needs to provision more warm containers to sustain the invocation requests for that function. If so, CSS re-enables the cold start path (line 11-12) and falls back to BSS (line 13-16). Otherwise, CSS determines that it is still worth just doing speculative wait without the need to enable the cold start path (line 17-18). All historical data, including T_i, T_e, T_p, and T_d, are collected using a 15-minute sliding window, whose size is configurable.

A novel take of our enhanced CSS policy is that it evaluates the probability that a cold start might be unnecessary using a simple, lightweight, hint-based classification. This method mitigates potential cache thrashing, as previously described, using a minimal set of metrics collected from historical executions. We evaluate the effectiveness of CSS in §5.1 and the impact of different historical sliding window sizes in §5.5.

3.3 Concurrency-Informed Priority

CIDRE evicts containers based on a new concurrency-informed priority (CIP) model. CIDRE assigns each cached warm container c a keep-alive priority. The priority is computed based on: (1) its *container-level statistics* including its reuse time, memory footprint, and the cold start latency, and (2) its *function-wise concurrency statistics* including the aggregate, function-wise invocation frequency, and the number of warm containers of that function (**Observation 2** in §2.4):

$$Priority = Clock(c) + Freq(\mathcal{F}(c)) \times \frac{Cost(c)}{Size(c) \times |\mathcal{F}(c)|} \quad (3)$$

Containers are sorted by priority which is updated during one of the following cases: (1) an idle warm container is used to execute the request, which is a *true warm start*, (2) a busy warm container is used to execute the request, in which case a queuing delay will incur and we call it a *delayed warm start*, and (3) a new container is provisioned and started due to insufficient resources, in which case some containers with the lowest priorities are reclaimed to release the resources.

Container-level Statistics. ***Size(c)*** and ***Cost(c)*** of a container c has the same definition as in FaasCache [27] (§2.4).

• ***Clock(c)*** captures the reuse recency of a container c. $Clock(c)$ is updated each time when c is invoked. When the cache is not full, newly created containers start with a clock value of 0. However, if CIDRE needs to evict some warm containers to make space for a new container c, c is assigned a clock value equal to the largest $Priority$ value among all evicted containers: $Clock(c) = \max_{e \in \text{Evicted}} Priority(e)$. This equation guarantees that the new container will always have a monotonically increasing clock value greater than those that are evicted, akin to the idea of a logical clock [35]. When a new request is served by a warm container, whether it is a true warm start or a delayed warm start, the clock value of the container is updated as the value of its current $Priority(c)$ before c's priority gets updated per Equation (3).

Function-level Statistics. Next, we discuss the intuition behind function-wise concurrency statistics. $\mathcal{F}(c)$ returns a set of all containers belonging to the same function as c.

• $|\mathcal{F}(c)|$ is the number of warm containers associated with $\mathcal{F}(c)$. The intuition is that caching excess warm containers for a function increases the likelihood of it occupying more resources than necessary.

• ***Freq(*$\mathcal{F}$*(c))*** computes the average number of invocations per minute for a function associated with $\mathcal{F}(c)$, providing an approximation of the average concurrency of a function:

$$Freq(\mathcal{F}(c)) = \frac{n_{\mathcal{F}(c)}}{t} \quad (4)$$

Where $n_{\mathcal{F}(c)}$ is the total number of invocations that the function associated with $\mathcal{F}(c)$ has ever received over its entire history, and t denotes the total duration in minutes since the first request of this function. Unlike traditional frequency-based caching policies that use the reuse count as object frequency (e.g., LFU and GDSF), $Freq(\mathcal{F}(c))$ measures a function's average invocation *rate* per minute. This method allows $Freq(\mathcal{F}(c))$ to adapt well to changing patterns since it can age stale containers with high reuse counts that may no longer be useful. If a function's warm containers are not being used for an extended period, the value of $Freq(\mathcal{F}(c))$ will decay as t increases while $n_{\mathcal{F}(c)}$ remains unchanged. Consequently, warm containers of this function may have a higher chance of eviction due to decreased priorities.

FaasCache's frequency captures the aggregate number of invocations across all warm containers for a function: a function with fewer warm containers tends to have a lower priority and is more likely to be evicted. CIDRE's function-level priority, in contrast, captures per-container frequency with a denominator $|\mathcal{F}(c)|$: with a fixed $Freq(\mathcal{F}(c))$, a smaller $\frac{Freq(\mathcal{F}(c))}{|\mathcal{F}(c)|}$ indicates that there are sufficient warm containers for this function. This suggests that retaining additional containers in the cache is unnecessary.

3.4 CIDRE: Putting It All Together

Putting it all together (Figure 11), we have the complete design of CIDRE container orchestration policy shown in Algorithm 2. CIDRE handles each arrived request targeting a function F in one of the following two cases.

- **Case I:** If there is an idle warm container in the cache, CIDRE simply dispatches the request to this warm container, resulting in a true warm start (best-case scenario). After the request is served, CIDRE updates the priority for the touched container (Subroutine UPDATE()).
- **Case II:** If the function cache does not have any available warm container to serve the request, CIDRE triggers CSS

Algorithm 2 CIDRE FaaS orchestration algorithm.

Input: Target function F. Priority queue PQ for all warm containers.
Case I. If an idle warm container is found, serve the request directly.
UPDATE(PQ, F). ▷ PQ updated asynchronously.
Case II. If no available warm containers can be found:
Algorithm 1.
REPLACE(PQ, F) and UPDATE(PQ, F). ▷ PQ updated asynchronously.
Subroutine UPDATE(PQ, F): ▷ Update priorities.
For reused container c of F, $Clock(c) = Priority(c)$.
For newly created container c of F,
$Clock(c) = \max_{e \in \text{Evicted}} Priority(e)$.
For each container c of F, $n_{\mathcal{F}(c)}$ ++ as in Eq. (4).
For newly created container c of F, $|\mathcal{F}(c)|$ increases by 1.
For each container c of each victim function associated with $\mathcal{F}(c)$ with $E_{\mathcal{F}(c)}$ evicted containers from PQ, $|\mathcal{F}(c)|$ decreases by $E_{\mathcal{F}(c)}$.
For each container c touched in previous steps, update its priority using Eq. (3).
Subroutine REPLACE(PQ, F): ▷ Perform container replacement.
For the newly created container c of F, compute
$E = \arg\min_{E'}(S_c \leq \sum_{i=1}^{E'} S_i)$, where S_c and S_i are the required memory size of cold-started container c and victim idle container S_i, respectively.
Evict E idle warm containers with the lowest priorities from PQ.
Create a new container of F.

(Algorithm 1). If CIDRE decides to provision a new container during CSS, a container replacement is triggered and Subroutine REPLACE() is called to evict E idle warm containers with lowest priorities. CIDRE then updates the priorities for all containers that are involved in this case.

Time Complexity. Since all the historical data used in Algorithm 1 and the PQ are collected and updated periodically and asynchronously, these steps are not on the critical path of request scheduling. The primary responsibility of Algorithm 1 is to decide when to turn on/off the BSS based on the collected historical data. The algorithm has a time complexity of $O(1)$, and our measurement indicates that Algorithm 1 introduces a negligible overhead of 36us.

4 Experimental Methodology

Implementation. We have implemented CIDRE in OpenLambda [44], an open-source FaaS platform written in Go. We implemented CSS in OpenLambda's worker and function management components. We added a new channel (a FIFO queue) in each function manager to buffer all requests that do not immediately find an available warm container. The implemented CSS strategy evaluates various metrics (§3.2) to determine the most cost-effective way to execute the next outstanding request from the head of the channel. CIDRE then pulls that request and takes action as informed by Algorithm 1. Compared to OpenLambda's existing policy, our CSS strategy is: (1) work-conserving as it serves requests using any vacant resources that become available the earliest, and (2) cost-effective as it conservatively stops provisioning new containers if there are sufficient resources to serve requests.

We implemented the concurrency-informed priority (CIP) cache eviction policy in OpenLambda's worker component by replacing its time-to-live keep-alive policy. CIDRE maintains updated container-level statistics within each container instance and updated function-level statistics within OpenLambda's function manager. For efficiency, CIDRE updates the PQ lazily: upon a cache eviction, the PQ is resorted based on the latest container- and function-level statistics.

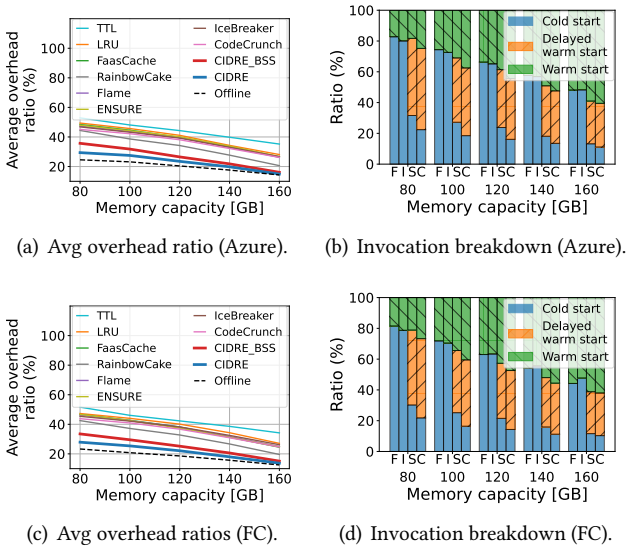
Production FaaS Traces. We sampled two large-scale FaaS workloads from the 30-minute Azure Functions and the 30-minute FC listed in Table 1. The new 30-minute Azure workload includes 330 sampled functions with around 598k invocation requests, and the new 30-minute FC workload includes 220 sampled functions with around 410k invocation requests. All the function apps used in the experiments are collected from two publicly available FaaS benchmarks [21, 33].

Compared Baselines. We compare CIDRE with a wide range of classic and SOTA baselines listed below:

- **TTL**: a time-to-live keep-alive policy that evicts containers based on container lifespan (10-minute expiration time), which is OpenLambda's default keep-alive policy.
- **LRU**: a least-recently-used (LRU) keep-alive policy that evicts containers based on recency.
- **FaasCache** [27]: an effective function keep-alive policy based on GDSF caching.
- **RainbowCake** [61]: a SOTA function pre-warming and keep-alive technique that warms up containers and keeps functions alive using layer-wise container sharing.
- **IceBreaker** [46]: a SOTA function pre-warming and keep-alive policy that exploits server heterogeneity to optimize the keep-alive cost.
- **CodeCrunch** [13]: a SOTA function keep-alive policy that exploits function compression and server heterogeneity to reduce the service time under high memory pressure.
- **Flame** [59]: a SOTA function keep-alive solution that uses a globally centralized cache manager for managing function caching.
- **ENSURE** [50]: a SOTA FaaS auto-scaler, which dynamically scales containers based on workload traffic to reduce cold starts and deactivates the unneeded containers to improve resource management.
- **Offline:** an offline CIDRE function orchestration policy, which utilizes future workload knowledge to make informed scaling and eviction decisions. Offline uses Belady's MIN [14] as its keep-alive policy, which evicts function containers that will be reused the furthest in the future. Offline makes informed scaling decisions by exhaustively searching all busy warm containers in the current and future cache state to find a container with the shortest waiting time; if the cold start cost is lower than a delayed warm start for all busy warm containers, Offline starts a new function container.

5 Evaluation

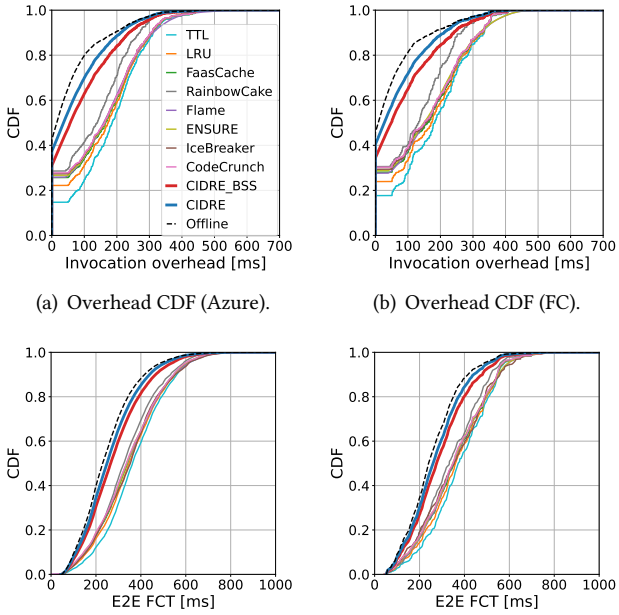
We evaluated our prototype CIDRE on a cluster of three servers, each with 64GB RAM and 64-core Intel CPUs running Ubuntu 22.04.1 LTS. Due to its effectiveness and simplicity, CIDRE_BSS has been deployed in Alibaba Cloud FC,



(a) Avg overhead ratio (Azure). (b) Invocation breakdown (Azure).

(c) Avg overhead ratios (FC). (d) Invocation breakdown (FC).

Figure 12. Comparison with a series of baselines for various cache sizes with a step of 20GB. In Figure 12(b) and 12(d), F: FaasCache, I: IceBreaker, S: CIDRE_BSS, C: CIDRE.



(a) Overhead CDF (Azure). (b) Overhead CDF (FC).

(c) E2E service time CDF (Azure). (d) E2E service time CDF (FC).

Figure 13. Invocation overhead and end-to-end service time of different systems with a 100GB cache.

serving 110k function invocation requests per second. To test CIDRE's effectiveness, we also evaluated a production-quality CIDRE_BSS deployed in production FC platform.

5.1 Baseline Comparison

Invocation Overhead. Figure 12(a) and 12(c) show that baseline policies spend a significant amount of time waiting for containers to be provisioned. CIDRE and CIDRE_BSS have a much smaller average overhead ratio than all the seven online baselines for all cache sizes, while the Offline achieves the highest efficiency. CIDRE maintains a similar improvement rate for other cache sizes.

CIDRE outperforms FaasCache and LRU (by up to 43.8% and 47.0% for average invocation overhead ratio, respectively), which both rely on caching-driven keep-alive for container eviction, by taking it a step further and speculatively selecting delayed warm starts under concurrency. RainbowCake uses fine-grained, layer-based pre-warming strategy to reduce the cold start cost. Pre-warming largely relies on future workload prediction for performance improvement. Compared to existing whole-container-based keep-alive policies (FaasCache and LRU), RainbowCake's layer-based warm-up strategy exposes higher chances of container sharing, thus having smaller invocation overhead. However, in highly concurrent workloads, the chances of finding enough available common layers in the cache get significantly reduced, and therefore, RainbowCake needs to either wait for a common layer to become available or create a new container, both of which incur a waiting time. Therefore, CIDRE achieves up to 33.7% lower average invocation overhead ratio compared to RainbowCake.

For an 100GB cache under the Azure workload, the invocation overhead ratio accounts for an average of 43.2% and 42.2% for IceBreaker and CodeCrunch, while CIDRE achieves an average invocation overhead ratio of 27.5%, a reduction of 36.3% and 34.8%, respectively. For the controlled experiment, we ran IceBreaker and CodeCrunch on the same three-node, homogeneous cluster. The homogeneous setting diminishes the potential benefit of IceBreaker's sophisticated optimizer. IceBreaker performs offline profiling to record the statistics about cold start costs, execution time, and memory usage of the entire workload and then runs optimization during real-time workload replay [46].

ENSURE proposes an autoscaling method (FnScale) that reserves additional capacity as "burst buffers" to handle bursts of workload demand. However, proactively reserving additional containers under high concurrency, especially with restricted global memory resources, can be challenging, thereby reducing the effectiveness of ENSURE. CIDRE demonstrates a performance improvement of up to 38.8% in average invocation overhead ratio compared to ENSURE. Flame exploits workload skewness by evicting rarely invoked cold functions but performs worse than CIDRE under high concurrency and high load.

Effectiveness of CSS. Figure 12(a) and 12(c) show that CIDRE with CSS achieves consistently lower average overhead ratio—reduced by 7.5%-17.6%—across all cache sizes compared to CIDRE_BSS with only basic SS enabled. BSS always creates a new container, even when opting for a delayed warm start, with the possibility that the new container may or may not be reused later. In contrast, CSS adopts a more conservative approach and chooses not to create a new container when the cache has sufficient warm containers to

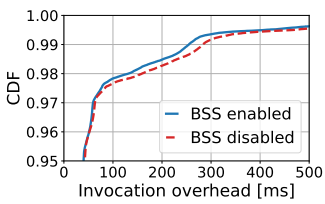


Figure 14. Invocation overhead in an FC production cluster.

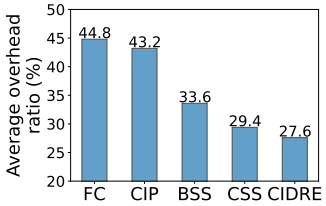


Figure 15. Ablation study of techniques in CIDRE.

handle new requests, thereby improving resource utilization and reducing the overhead. Thus, CIDRE reduces the number of (wasted) cold starts compared to CIDRE_BSS (Figure 12(b) and 12(d)), leading to more efficient use of the limited cache space with a higher warm start ratio than CIDRE_BSS.

Cold Start Ratio. As shown in Figure 12(b) and 12(d), CIDRE and CIDRE_BSS have dramatically smaller cold start ratios compared to FaasCache and IceBreaker. By speculatively waiting for busy warm containers and executing requests on them, CIDRE and CIDRE_BSS effectively convert an enormous amount of cold starts into delayed warm starts. For example, CIDRE reduces the cold start ratio of FaasCache by 75.1% for a 100GB cache under the Azure workload.

End-to-End Service Time. Figure 13(c) and 13(d) show how CIDRE and CIDRE_BSS help with improving the end-to-end service time. Service time measures the time span from the arrival of the request to the completion of the request. By minimizing the invocation overhead, CIDRE and CIDRE_BSS reduce the E2E service time as well, with CIDRE approaching the best-case baseline Offline. CIDRE, FaasCache, and CodeCrunch have a 50^{th}-ile (90^{th}-ile) E2E service time of: 249.76 ms (438.32 ms), 342.23 ms (548.89 ms), and 330.50 ms (542.43 ms) under the Azure workload, respectively.

5.2 CIDRE in Alibaba Cloud FC Production Cluster

We tested CIDRE in a production FC cluster by toggling the BSS setting on and off. The workload consists of around 410k invocation requests sampled from the FC trace (Table 1). The production cluster contains *37 bare-metal machines, each having 384GB RAM and 104 CPUs, hosting 1,500 function container instances, sharing a global resource pool* with other FC FaaS tenants. It follows the same production configurations as the overall platform, *ensuring low latency and high SLO.*

The test exhibited a cold start ratio of 1.10% with BSS disabled, consistent with production cold start statistics reported in Flame [59]. Enabling BSS helps reduce the cold start ratio by 34.5%, bringing it down to 0.72%. As shown in Figure 14, BSS reduces the 99^{th}-ile invocation overhead (254.67 ms) by 10.01% compared to when BSS is disabled (283 ms). This result demonstrates that CIDRE is simple and effective, and is easily deployable to already sophisticated production systems.

5.3 Ablation Study

Figure 15 shows the contributions of each technique of CIDRE in reducing the invocation overhead with 100 GB cache. We tested the following three CIDRE configurations: (1) CSS_alone: CIDRE with CSS enabled and CIP disabled, (2) BSS_alone: CIDRE with basic SS enabled and CIP disabled, (3) CIP_alone: CIDRE with CIP enabled and BSS/CSS disabled. CIP_alone reduces the overhead of FaasCache by 3.6% due to more balanced concurrency-informed evictions across all functions. With a basic SS strategy, BSS_alone sees a huge improvement in average overhead compared to CIP_alone, thanks to more efficient use of existing warm containers. CSS_alone further reduces the average overhead by 12.5% compared to BSS_alone. With both CSS and CIP enabled, CIDRE exhibits a 6.1% reduction in average overhead compared to CSS_alone, demonstrating the efficacy of CIDRE's overall concurrency-informed orchestration strategy.

5.4 Concurrency-Driven Scaling

In this test, we varied the average level of concurrency and measured the corresponding average memory resource usage in gigabytes (GB) given a concurrency level. Figure 16 shows the results with a 100GB cache. The memory usage, i.e., the number of containers created, increases as the concurrency level scales out across all four systems tested. CIDRE_BSS's basic SS policy has lower memory usage at all concurrency levels when compared to FaasCache, since CIDRE_BSS's speculative waiting reduces the cold start ratio. As the concurrency level increases, this gap becomes smaller as excessive cold starts cause cache thrashing. CIDRE requires the least number of containers to sustain a burst of concurrent requests, with a saving of up to 22% compared to FaasCache, under the highest concurrency level. The reduction in created containers is because CIDRE disables the cold start path when it detects potential cache thrashing where provisioning a new container would cause the eviction of an existing warm container that will be reused soon. With a more conservative cold start control, CIDRE achieves a lower cold start ratio than both FaasCache and CIDRE_BSS.

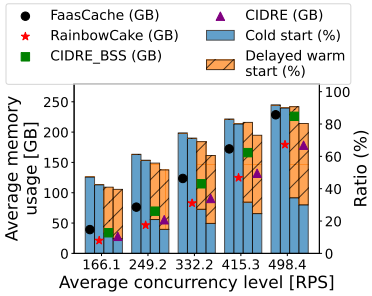


Figure 16. Concurrency-driven scaling. RPS: requests per second.

RainbowCake uses the least memory for handling concurrent requests when the average concurrency level is below 498.4, achieving memory savings of 24.7% and 5.6% at concurrency levels of 166.1 and 415.3, respectively, compared to CIDRE. However, this comes with significantly higher

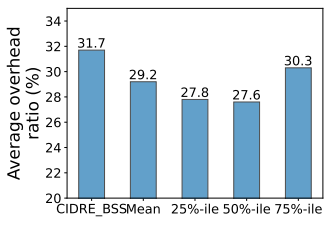


Figure 17. Average invocation overhead ratio with different execution time thresholds T_e.

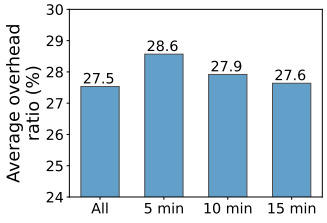


Figure 18. Average invocation overhead ratio in varying the historical data window.

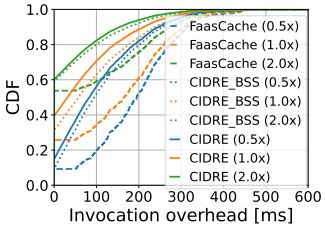


Figure 19. CDFs of invocation overhead with different IAT levels.

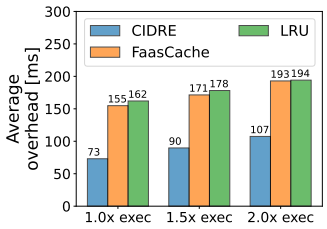


Figure 20. Impact of varying the function execution time on invocation overhead.

cold start ratios, demonstrating an *interesting tradeoff between performance (cold start ratio or invocation overhead) and memory usage*. At the highest concurrency level, RainbowCake shows only 0.5% more memory usage than CIDRE. At lower concurrency levels, RainbowCake benefits from sufficient common layers in the cache for sharing, minimizing memory requirements for container layers. However, as concurrency increases, incoming requests may not find idle common layers available, leading to the creation of additional containers and higher memory consumption.

The results can be explained from the classic spatial vs. temporal locality perspective. RainbowCake introduces an intermediate caching state at the *memory space dimension*, sharing fine-grained layers to reduce overall memory cost and startup overhead. However, under high concurrency, new layers or containers must be provisioned when no idle or shareable layers are available, increasing memory usage. In contrast, CIDRE introduces a new caching state that operates in the *temporal dimension*, strategically waiting for a "delayed hit" beyond traditional cache hit/miss decisions.

5.5 Sensitivity Analysis

Thus far we have focused on default configurations. In this section we perform a sensitivity study to understand the impact of various configurations on invocation overhead. All experiments in this section were conducted using the Azure workload with a 100 GB cache.

Estimated Execution Time Threshold. We first study the impact of different estimated execution time threshold T_e on invocation overhead. We tested different configurations (mean, 25^{th}%-ile, 50^{th}%-ile, and 75^{th}%-ile) of the historical execution time used in CSS (Algorithm 1). Figure 17 plots the invocation overhead ratio for the Azure workload. CIDRE Mean and CIDRE 75^{th}%-ile perform better than CIDRE_BSS but worse than CIDRE 50^{th}%-ile. This result suggests that a 25^{th}%-ile threshold might be a little small while a 75^{th}%-ile could be too large. Therefore, we empirically selected the 50^{th}%-ile as T_e throughout our evaluation.

Historical Sliding Window Sizes. We evaluate how varying the amount of historical data impacts the invocation overhead for CSS. To do this, we examined different sliding window lengths (all, 5 minutes, 10 minutes, and 15 minutes) for collecting historical data for CSS as outlined in Algorithm 1. Figure 18 shows the invocation overhead ratio for each time window. Collecting all available historical data results in the lowest overhead ratio. Using 10-minute and 15-minute time windows slightly underperforms compared to the all-data configuration. The results suggest that the 10-minute and 15-minute windows yield reasonably acceptable performance. Therefore, we chose a 15-minute sliding window for historical data collection throughout the evaluation.

Table 2. Sensitivity experiment for different execution times. CR: cold start ratio (%). WR: warm start ratio (%). DR: delayed warm start ratio (%).

Method	CR (1.0× / 1.5× / 2.0×)	WR (1.0× / 1.5× / 2.0×)	DR (1.0× / 1.5× / 2.0×)
CIDRE	18.5 / 22.3 / 27.6	37.5 / 22.1 / 8.5	44.0 / 55.6 / 63.9
FaasCache	74.4 / 82.2 / 92.6	25.6 / 17.8 / 7.4	N/A
LRU	78.3 / 85.5 / 93.2	21.7 / 14.5 / 6.8	N/A

Inter-Arrival Times. Next, Figure 19 shows the impact of different IAT levels on invocation overhead. We varied the IAT factor from 0.5× to 2×, where 1× means the original workload's IAT behavior, and a longer (shorter) IAT means a lower (higher) average load. As the load increases (reduced IAT), the invocation overhead increases and the warm start ratio reduces. CIDRE achieves a warm start ratio of 60.4%, 39.5%, and 15.0% under the IAT level of 2×, 1×, and 0.5×, respectively. However, CIDRE's performance benefit holds consistently against other baselines across all IAT levels.

Function Execution Time. Next, we explore how different execution times impact the invocation overhead. We varied the inputs for each function to adjust the execution time to 1.0×, 1.5×, and 2.0× of the original execution time. For this analysis, we present performance metrics in terms of invocation overhead rather than invocation overhead ratios, which can be influenced by execution time. Figure 20 and Table 2 show the invocation overheads and their breakdowns for each execution time. As the execution time increases, the likelihood of incoming requests finding an idle container for a warm start decreases. This results in a higher cold start ratio (Table 2) and an increase in average invocation overhead (Figure 20). With delayed warm starts for CIDRE, 70.4%, 71.4%, and 69.9% of non-warm starts were executed as delayed warm starts for execution times of 1.0×, 1.5×, and 2.0×, respectively. These results are consistent with the analysis of the delayed warm start opportunity space shown

in Figure 10 (§2), indicating that while varied execution times may shift completion times, they do not fundamentally alter the overall distribution of delayed warm start opportunities.

Number of Intra-Container Threads. Finally, we examined the impact of varying number of intra-container threads. An N-thread function container is capable of handling N simultaneous requests, and a new container will only be provisioned if the maximum allowable number of threads N or the maximum memory limit is reached. We compared FaasCache and CIDRE with function containers configured for 1 to 8 threads. The 1-thread configuration serves as the baseline (our default setting), representing the method in which each container processes only one request at a time.

Figure 21 shows the invocation overhead ratio for FaasCache and CIDRE across various thread configurations. As the number of threads increases, both FaasCache and CIDRE show a decrease in the average overhead ratio. By allowing multiple requests to be processed in parallel within the same container, a larger number of requests can utilize available CPU resources for execution as warm starts, leading to a significant reduction in the cold start ratio and the average overhead ratio. Although CIDRE chooses a delayed warm start only when containers reach their maximum thread capacity, it consistently achieves a lower average overhead ratio compared to FaasCache across all thread numbers. This enhancement occurs because CIDRE effectively minimizes invocation overheads by enabling more cold starts to be executed as delayed warm starts with reduced latency. This result demonstrates that CIDRE's speculative scaling remains effective even when function containers are equipped with additional CPU power.

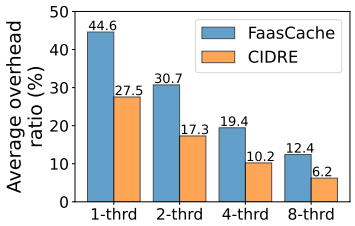


Figure 21. Impact of varying the number of intra-container threads.

6 Related Work

Mitigating Cold Start Costs. A line of work focuses on optimizing the cold start costs of serverless functions [4, 5, 7, 23, 27, 40, 41, 46, 49, 50, 52, 56]. A common practice to tackle cold start penalty is to cache provisioned function sandboxes in memory [51, 55]. RainbowCake [61] shares container layers from idle warm containers to speed up the cold start. Icebreaker [46] exploited heterogeneous function hosts to reduce the keep-alive cost. SAND [5] and Pagurus [36] share and reuse container runtimes to alleviate cold starts. Researchers proposed to use snapshot loading [7, 16, 17, 23, 52] to jump start function cold start from disk images. CIDRE exploits a new tradeoff and optimizes function scaling and eviction spanning the entire serverless function lifecycle.

Latency-aware Caching. Atre et al. [9] found that traditional caching policies fail to minimize latency in the presence of delayed hits, where under high throughput, multiple I/O requests to the same object queue up before an outstanding cache miss is resolved [58]. Delayed warm starts in FaaS may seem similar to delayed I/O hits as both involve delayed accesses to cached objects. However, they are fundamentally different: (1) **Root causes and contexts**: Delayed hits stem from the slow process of loading missed objects from a backing store into the cache, while delayed warm starts are caused by FaaS concurrency. (2) **Impacts**: Delayed hits cause latency increase for subsequent "hits" accumulated in the queue, while delayed warm starts present opportunities for reducing the invocation latency.

Delay Scheduling. Zaharia et al. [62] proposed delay scheduling, which addresses the tension between fair-sharing scheduling and data locality for traditional MapReduce cluster computing workloads. Enforcing a task to wait for a limited time on a busy slot could expose better data locality. CIDRE's speculative scaling is similar in that CIDRE imposes a delay when scheduling serverless function requests for "better container locality". CIDRE's speculative scaling is different from delay scheduling in that it tackles new challenges of how to balance cold starts vs. delayed warm starts to avoid cache thrashing and resource wastage in a novel context of highly concurrent FaaS workloads.

7 Conclusion

The key insight of this paper is that function keep-alive should be optimized for FaaS concurrency behavior. We identify a new tradeoff between delayed warm starts and cold starts. Intelligently reusing busy warm containers not only reduces the latency but also reduces cold starts, leading to improved warm start ratios and cost-effective function scaling. We also find that function keep-alive must consider both container-level statistics and function-level concurrency to inform eviction decisions. We built CIDRE atop OpenLambda and evaluated it using production workloads. Results show that CIDRE significantly outperforms state-of-the-art FaaS keep-alive solutions. CIDRE's speculative scaling policy has been adopted by Alibaba Cloud Function Compute. Deploying CSS in production is part of our future work.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Kostis Kaffes, for their valuable feedback and comments. This research was supported in part by U.S. NSF grants NSF-2350425, NSF-2319988, NSF-2206522, NSF-2322860, NSF-2318628, NSF CloudBank, Microsoft Research Faculty Fellowship 8300751, Amazon research award, AWS Cloud Credit for Research, and the Commonwealth Cyber Initiative (CCI), an investment in the advancement of cyber research, innovation and workforce development. For more information about CCI, visit cyberinitiative.org.

References

[1] Alibaba Cloud Function Compute. https://www.alibabacloud.com/product/function-compute.

[2] Knative: An Open-Source Enterprise-level solution to build Serverless and Event Driven Applications. https://knative.dev/docs/.

[3] Dimitris Achlioptas, Marek Chrobak, and John Noga. Competitive analysis of randomized paging algorithms. *Theoretical Computer Science*, 234(1-2):203–218, 2000.

[4] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX symposium on networked systems design and implementation (NSDI 20)*, pages 419–434, 2020.

[5] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. {SAND}: Towards {High-Performance} serverless computing. In *2018 Usenix Annual Technical Conference (USENIX ATC 18)*, pages 923–935, 2018.

[6] Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. Sprocket: A serverless video processing framework. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, page 263–274, New York, NY, USA, 2018. Association for Computing Machinery.

[7] Lixiang Ao, George Porter, and Geoffrey M. Voelker. Faasnap: Faas made fast using snapshot-based vms. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 730–746, New York, NY, USA, 2022. Association for Computing Machinery.

[8] Apache OpenWhisk. https://openwhisk.apache.org/, 2016.

[9] Nirav Atre, Justine Sherry, Weina Wang, and Daniel S. Berger. Caching with delayed hits. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 495–513, New York, NY, USA, 2020. Association for Computing Machinery.

[10] AWS Lambda. https://aws.amazon.com/lambda/, 2014.

[11] Azure Functions. https://azure.microsoft.com/en-us/services/functions/, 2019.

[12] Sorav Bansal and Dharmendra S. Modha. CAR: Clock with adaptive replacement. In *3rd USENIX Conference on File and Storage Technologies (FAST 04)*, San Francisco, CA, March 2004. USENIX Association.

[13] Rohan Basu Roy, Tirthak Patel, Rohan Garg, and Devesh Tiwari. Codecrunch: Improving serverless performance via function compression and cost-aware warmup location optimization. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 85–101, 2024.

[14] Laszlo A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal*, 5(2):78–101, 1966.

[15] Vivek M Bhasi, Jashwant Raj Gunasekaran, Aakash Sharma, Mahmut Taylan Kandemir, and Chita Das. Cypress: Input size-sensitive container provisioning and request scheduling for serverless platforms. In *Proceedings of the 13th Symposium on Cloud Computing*, pages 257–272, 2022.

[16] Marc Brooker, Mike Danilov, Chris Greenwood, and Phil Piwonka. On-demand container loading in AWS lambda. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 315–328, Boston, MA, July 2023. USENIX Association.

[17] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. Seuss: Skip redundant paths to make serverless fast. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[18] Benjamin Carver, Runzhou Han, Jingyuan Zhang, Mai Zheng, and Yue Cheng. λfs: A scalable and elastic distributed file system metadata service using serverless functions. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*, ASPLOS '23, page 394–411, New York, NY, USA, 2024. Association for Computing Machinery.

[19] Benjamin Carver, Jingyuan Zhang, Ao Wang, Ali Anwar, Panruo Wu, and Yue Cheng. Wukong: a scalable and locality-enhanced framework for serverless parallel computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 1–15, New York, NY, USA, 2020. Association for Computing Machinery.

[20] Ludmila Cherkasova. *Improving WWW proxies performance with greedy-dual-size-frequency caching policy*. Hewlett-Packard Laboratories Palo Alto, CA, 1998.

[21] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. Sebs: A serverless benchmark suite for function-as-a-service computing. In *Proceedings of the 22nd International Middleware Conference*, pages 64–78, 2021.

[22] Docker: Accelerated Container Application Development. https://www.docker.com/, 2013.

[23] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 467–481, 2020.

[24] Simon Eismann, Long Bui, Johannes Grohmann, Cristina Abad, Nikolas Herbst, and Samuel Kounev. Sizeless: Predicting the optimal size of serverless functions. In *Proceedings of the 22nd International Middleware Conference*, pages 248–259, 2021.

[25] Amos Fiat, Richard M Karp, Michael Luby, Lyle A McGeoch, Daniel D Sleator, and Neal E Young. Competitive paging algorithms. *Journal of Algorithms*, 12(4):685–699, 1991.

[26] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, fast and slow: Low-Latency video processing using thousands of tiny threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 363–376, Boston, MA, March 2017. USENIX Association.

[27] Alexander Fuerst and Prateek Sharma. Faascache: keeping serverless computing alive with greedy-dual caching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 386–400, 2021.

[28] Jim Gray. Why do computers stop and what can be done about it?, 1985.

[29] gVisor: The Container Security Platform. https://gvisor.dev/, 2018.

[30] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Serverless computation with OpenLambda. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, Denver, CO, June 2016. USENIX Association.

[31] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the cloud: Distributed computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 445–451, New York, NY, USA, 2017. Association for Computing Machinery.

[32] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383*, 2019.

[33] Jeongchul Kim and Kyungyong Lee. Functionbench: A suite of workloads for serverless cloud function service. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 502–504. IEEE, 2019.

[34] Lambda function scaling. https://docs.aws.amazon.com/lambda/latest/dg/lambda-concurrency.html, 2014.

[35] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, jul 1978.

[36] Zijun Li, Linsong Guo, Quan Chen, Jiagan Cheng, Chuhao Xu, Deze Zeng, Zhuo Song, Tao Ma, Yong Yang, Chao Li, and Minyi Guo. Help rather than recycle: Alleviating cold startup in serverless computing through Inter-Function container sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 69–84, Carlsbad, CA, July 2022. USENIX Association.

[37] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Eshaan Minocha, Sameh Elnikety, Saurabh Bagchi, and Somali Chaterji. Wisefuse: Workload characterization and dag transformation for serverless workflows. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 6(2):1–28, 2022.

[38] Lyle A McGeoch and Daniel D Sleator. A strongly competitive randomized paging algorithm. *Algorithmica*, 6(1-6):816–825, 1991.

[39] Nimrod Megiddo and Dharmendra S. Modha. ARC: A Self-Tuning, low overhead replacement cache. In *2nd USENIX Conference on File and Storage Technologies (FAST 03)*, San Francisco, CA, March 2003. USENIX Association.

[40] Anup Mohan, Harshad Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. Agile cold starts for scalable serverless. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, 2019.

[41] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. {SOCK}: Rapid task provisioning with {Serverless-Optimized} containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 57–70, 2018.

[42] OpenFaaS: Dude where's my coldstart. https://www.openfaas.com/blog/what-serverless-coldstart/, 2022.

[43] OpenFaaS: Server Functions, Made Simple. https://www.openfaas.com, 2016.

[44] OpenLambda. https://github.com/open-lambda, 2016.

[45] Apache OpenWhisk. Open source serverless cloud platform. *Executes functions in response to events at any scale*, 2020.

[46] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. Icebreaker: warming serverless functions better with heterogeneity. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 753–767, 2022.

[47] Gor Safaryan, Anshul Jindal, Mohak Chadha, and Michael Gerndt. Slam: Slo-aware memory optimization for serverless applications. In *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*, pages 30–39. IEEE, 2022.

[48] Serverless image handler. https://aws.amazon.com/solutions/implementations/serverless-image-handler/, 2014.

[49] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 205–218. USENIX Association, July 2020.

[50] Amoghavarsha Suresh, Gagan Somashekar, Anandh Varadarajan, Veerendra Ramesh Kakarla, Hima Upadhyay, and Anshul Gandhi. Ensure: Efficient scheduling and autonomous resource management in serverless environments. In *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, pages 1–10. IEEE, 2020.

[51] Understanding Container Reuse in AWS Lambda. https://aws.amazon.com/blogs/compute/container-reuse-in-lambda/, 2014.

[52] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. Benchmarking, analysis, and optimization of serverless function snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 559–572, New York, NY, USA, 2021. Association for Computing Machinery.

[53] Ao Wang, Shuai Chang, Huangshi Tian, Hongqi Wang, Haoran Yang, Huiba Li, Rui Du, and Yue Cheng. {FaaSNet}: Scalable and fast provisioning of custom serverless container runtimes at alibaba cloud function compute. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 443–457, 2021.

[54] Ao Wang, Jingyuan Zhang, Xiaolong Ma, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Vasily Tarasov, Feng Yan, and Yue Cheng. InfiniCache: Exploiting ephemeral serverless functions to build a Cost-Effective memory cache. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 267–281, Santa Clara, CA, February 2020. USENIX Association.

[55] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 133–146, Boston, MA, July 2018. USENIX Association.

[56] Xingda Wei, Tianxia Wang, Jinyu Gu, Yuhan Yang, Fangming Lu, Rong Chen, and Haibo Chen. Booting 10k serverless functions within one second via rdma-based remote fork. *arXiv preprint arXiv:2203.10225*, 2022.

[57] Working with Lambda container images. https://docs.aws.amazon.com/lambda/latest/dg/images-create.html, 2014.

[58] Gang Yan and Jian Li. Towards latency awareness for content delivery network caching. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 789–804, Carlsbad, CA, July 2022. USENIX Association.

[59] Yanan Yang, Laiping Zhao, Yiming Li, Shihao Wu, Yuechan Hao, Yuchi Ma, and Keqiu Li. Flame: A centralized cache controller for serverless computing. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*, pages 153–168, 2023.

[60] Hanfei Yu, Christian Fontenot, Hao Wang, Jian Li, Xu Yuan, and Seung-Jong Park. Libra: Harvesting idle resources safely and timely in serverless clusters. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '23, page 181–194, New York, NY, USA, 2023. Association for Computing Machinery.

[61] Hanfei Yu, Rohan Basu Roy, Christian Fontenot, Devesh Tiwari, Jian Li, Hong Zhang, Hao Wang, and Seung-Jong Park. Rainbowcake: Mitigating cold-starts in serverless with layer-wise container caching and sharing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2024.

[62] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems*, pages 265–278, 2010.

[63] Jingyuan Zhang, Ao Wang, Xiaolong Ma, Benjamin Carver, Nicholas John Newman, Ali Anwar, Lukas Rupprecht, Vasily Tarasov, Dimitrios Skourtis, Feng Yan, and Yue Cheng. Infinistore: Elastic serverless cloud storage. *Proc. VLDB Endow.*, 16(7):1629–1642, March 2023.